Introduction to Python for Data Scientists

Agenda

- Python Basics
- Shell & Magic commands
- Functions
- Control flow
- Handling Files
- Scalar types (Numeric, String, Boolean, None)
- Data structures (Tuple, List, Dict, Set)
- List, Set, and Dict Comprehensions

- Annex: Python notebook

How to start python?

- Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time.
- The standard interactive Python interpreter can be invoked on the command line with the python command.
- To exit the Python interpreter and return to the command prompt, you can either type exit() or press Ctrl-D.
- Running Python programs is as simple as calling python with a .py file as its first argument.

```
dsc: python_class % python
Python 2.7.11 |Anaconda 2.5.0 (64-bit)| (default, Dec  6 2015, 18:08:32)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> a=5
>>> print a
5
>>> print a*2
10
>>>
dsc: python_class % echo "print 'Hello world from python!'" > hello_world.py
dsc: python_class % python hello_world.py
Hello world from python!
```

How to start python?

- While many Python programmers execute all of their Python code in this way, many *scientific* Python programmers make use of IPython, an enhanced interactive Python interpreter.
- Ipython comes with command history and auto completion and interactivity
- Fernando Pérez, a physics grad student https://www.youtube.com/watch?v=g8xQRI3E8r8
- By using the **%run** command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done.

```
dsc: python_class % ipython
Python 2.7.11 |Anaconda 2.5.0 (64-bit)| (default, Dec  6 2015, 18:08:32)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.3 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world from python!

In [2]:
```

The Basics

- The Python language design is distinguished by its emphasis on readability, simplicity, and explicitness.
- Some people go so far as to liken it to “executable pseudocode”.
- Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

- A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block.
- One major reason that whitespace matters is that it results in most Python code looking cosmetically similar, which means less cognitive dissonance when you read a piece of code that you didn’t write yourself (or wrote in a hurry a year ago!).
- Some people use tabs or a different number of spaces, with 2 spaces not being terribly uncommon. 4 spaces is by and large the standard adopted by the vast majority of Python programmers

The Basics

- Statements do not need to be terminated by semicolons.
  - Semicolons can be used, however, to separate multiple statements on a single line (a = 5; b = 6; c = 7)
- Everything is an object
  - Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own “box” which is referred to as a Python object.
  - Each object has an associated type (for example, string or function) and internal data.
  - In practice this makes the language very flexible, as even functions can be treated just like any other object.
- Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter
  - For multi-line comments use triple-quoted strings (‘’’ text ‘’’)
- Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:
  - result = f(x, y, z)
  - g( )
- Almost every object in Python has attached functions, known as methods, that have access to the object’s internal contents
  - obj.some_method(x, y, z)

The Basics

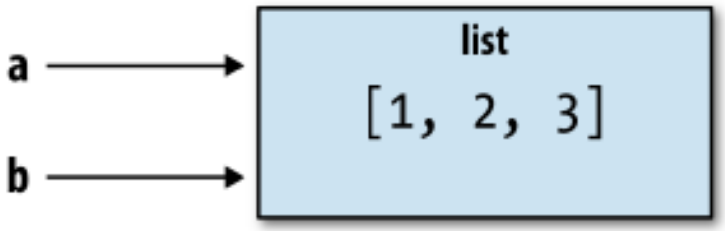
- When assigning a variable (or *name*) in Python, you are creating a *reference* to the object on the right hand side of the equals sign.

```
In [1]: a=[1,2,3]

In [2]: b=a

In [3]: a.append(4)

In [4]: b
Out[4]: [1, 2, 3, 4]
```



- In some languages, a=b would cause the data [1, 2, 3] to be copied. In Python, a and b actually now refer to the same object, the original list [1, 2, 3].
- When you pass objects as arguments to a function, you are only passing references; no copying occurs.
- Thus, Python is said to pass by reference, whereas some other languages support both pass by value (creating copies) and pass by reference.
- This means that a function can mutate the internals of its arguments.

The Basics - Objects in Python

- object references in Python have no type associated with them.
- Variables are names for objects within a particular namespace; the type information is stored in the object itself.

```
In [20]: a=5

In [21]: type(5)
Out[21]: int

In [22]: a='foo'; type(a)
Out[22]: str
```

- Python is considered a strongly-typed language, which means that every object has a specific type (or class), and implicit conversions will occur only in certain obvious circumstances.

```
In [23]: '5'+5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-8b3dc1ff685c> in <module>()
----> 1 '5'+5

TypeError: cannot concatenate 'str' and 'int' objects

In [24]: a=5.5; b=2

In [25]: a/b
Out[25]: 2.75
```

The Basics - Objects in Python

- Objects in Python typically have both attributes, other Python objects stored “inside” the object, and methods, functions associated with an object which can have access to the object’s internal data.
- Both of them are accessed via the syntax obj.attribute_name:

```
In [35]: a= "text"

In [36]: a.
a.capitalize  a.format      a.isupper     a.rindex      a.strip
a.center      a.index       a.join        a.rjust       a.swapcase
a.count       a.isalnum     a.ljust       a.rpartition  a.title
a.decode      a.isalpha     a.lower       a.rsplit      a.translate
a.encode      a.isdigit     a.lstrip      a.rstrip      a.upper
a.endswith    a.islower     a.partition   a.split       a.zfill
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

The Basics - Duck typing

- Python supports Duck typing -> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck
- Duck typing means that you don't care about the type of an object but rather only whether it has certain methods or behavior.

```
In [39]: def isiterable(obj):
    ....:     try:
    ....:         iter(obj)
    ....:         return True
    ....:     except TypeError: # not iterable
    ....:         return False
    ....:

In [40]: isiterable('a string')
Out[40]: True

In [41]: isiterable([1, 2, 3])
Out[41]: True

In [42]: isiterable(5)
Out[42]: False
```

The Basics - Strictness versus laziness

Python is a very strict (or eager) language. Nearly all of the time, computations and expressions are evaluated immediately.

```
In [4]: a = b = c = 5

In [5]: d = a + b * c

In [6]: d
Out[6]: 30
```

- In Python, once these statements are evaluated, the calculation is immediately (or strictly) carried out, setting the value of d to 30. In the above simple expression, the result of b * c is computed as a separate step before adding it to a.

- In another programming paradigm, such as in a pure functional programming language like Haskell, the value of d might not be evaluated until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as lazy evaluation

The Basics - Mutable and immutable objects

- Most objects in Python are mutable, such as lists, dicts, NumPy arrays, or most userdefined types (classes). This means that the object or values that they contain can be modified.

```
In [1]: a_list = ['foo', 2, [4, 5]]

In [2]: a_list[2] = (3, 4)

In [3]: a_list
Out[3]: ['foo', 2, (3, 4)]
```

- Others, like strings and tuples, are immutable:

```
In [7]: a_tuple = (3, 5, (4, 5))

In [8]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'

TypeError: 'tuple' object does not support item assignment
```

The Basics - imports

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

- in Python a module is simply a .py file containing function and variable definitions
- to access the variables and functions defined:

```
In [14]: import some_module

In [15]: result=some_module.f(5)

In [16]: pi=some_module.PI

In [17]: print "result=",result,", pi=",pi
result= 7 , pi= 3.14159
```

- Or equivalently:

```
In [24]: from some_module import f, g as just_another_funct, PI

In [25]: result = just_another_funct(5, PI)

In [26]: print result
8.14159
```

The shell commands

- We have shell with ! command
- Tab also works
- IMPORTANT!! The shell where !command runs is immediately discarded after executing 'command'.

```
In [40]: ! pwd
/home/dsc/python_class

In [41]: ! ls -l
total 180
-rw-rw-r--. 1 dsc dsc    95 Apr  4 12:35 hello_world.py
-rw-rw-r--. 1 dsc dsc   212 Apr  6 08:09 say_hello.py.py
-rw-rw-r--. 1 dsc dsc    84 Apr  4 14:34 some_module.py
-rw-rw-r--. 1 dsc dsc   362 Apr  4 14:36 some_module.pyc
-rw-rw-r--. 1 dsc dsc 43675 Apr  6 10:15 test01.ipynb
-rw-rw-r--. 1 dsc dsc  5422 Apr  5 20:17 test03.ipynb
-rw-rw-r--. 1 dsc dsc 89812 Apr  5 22:50 test2.ipynb
-rw-rw-r--. 1 dsc dsc 20208 Apr  5 09:31 test2.ipynb.01
-rw-rw-r--. 1 dsc dsc     0 Apr  5 13:17 test.txt
-rw-rw-r--. 1 dsc dsc   582 Apr  5 22:23 Untitled.ipynb

In [42]: ! cat h
%%html          %history        hash            help
%hist           hasattr         hello_world.py  hex

In [42]: ! cat hello_world.py
```

The shell commands

```
In [42]: ! grep -i "hello" hello_world.py
print 'Hello world from python!'

In [43]: ! grep "hello" hello_world.py

In [44]: ! psql -d optd
psql (9.4.6)
Type "help" for help.

optd=# \d
                    List of relations
 Schema |               Name                | Type  | Owner
--------+-----------------------------------+-------+-------
 public | airports                          | table | dsc
 public | continents                        | table | dsc
```

- Run python from within python

```
In [46]: !python hello_world.py
Hello world from python!

In [47]: %run hello_world.py
Hello world from python!
```

The shell commands

- Try cd command!

It fails silently….. Why doesn't it work?

```
In [61]: ! pwd
/home/dsc/python_class

In [62]: ! cd ..

In [63]: ! pwd
/home/dsc/python_class

In [64]: %cd -
/home/dsc

In [65]: ! pwd
/home/dsc
```

What did we use instead?

The Magic commands

- IPython has a set of predefined ‘magic functions’ that you can call with a command line style syntax. There are two kinds of magics, line-oriented and cell-oriented.
- Line magics are prefixed with the % character and work much like OS command-line calls: they get as an argument the rest of the line, where arguments are passed without parentheses or quotes.
- Cell magics are prefixed with a double %%, and they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument.

```
In [10]: %lsmagic
Out[10]:
Available line magics:
%alias  %alias_magic  %autocall  %autoindent  %automagic  %bookmark  %cat  %cd  %clear
%colors  %config  %cp  %cpaste  %debug  %dhist  %dirs  %doctest_mode  %ed  %edit  %env
%gui  %hist  %history  %install_default_config  %install_ext  %install_profiles  %killbg
scripts  %ldir  %less  %lf  %lk  %ll  %load  %load_ext  %loadpy  %logoff  %logon  %logst
art  %logstate  %logstop  %ls  %lsmagic  %lx  %macro  %magic  %man  %matplotlib  %mkdir
 %more  %mv  %notebook  %page  %paste  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %p
info2  %popd  %pprint  %precision  %profile  %prun  %psearch  %psource  %pushd  %pwd  %p
ycat  %pylab  %quickref  %recall  %rehashx  %reload_ext  %rep  %rerun  %reset  %reset_se
lective  %rm  %rmdir  %run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %tim
eit  %unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%latex  %
%perl  %%prun  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%
sx  %%system  %%time  %%timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Line Magic commands

- %automagic = Make magic functions callable without having to type the initial %.
- %cd = Change the current working directory
- %dhist = Print your history of visited directories.

- %run -Run the named file inside IPython as a program.
- %quickref = Show a quick reference cheat sheet

- %env = Get, set, or list environment variables.
- %set_env = Set environment variables

- %matplotlib = Set up matplotlib to work interactively.
    - %matplotlib inline
- %precision : Set floating point precision for pretty printing.

```
In [9]: 10/3.
Out[9]: 3.3333333333333335

In [10]: %precision 3
Out[10]: u'%.3f'

In [11]: 10/3.
Out[11]: 3.333
```

Line Magic commands

- %config = configure ipython
  - To see what classes are available for config, pass no arguments
  - To view what is configurable on a given class, just pass the class name:
  - To view one parameter pass class_name.parameter
  - To change the parameter: config TerminalInteractiveShell.editor='gedit'

```
In [16]: config
Available objects for config:
     TerminalInteractiveShell
     HistoryManager
     PrefilterManager
     IPCompleter
     PromptManager
     DisplayFormatter
     MagicsManager
     ScriptMagics
     AliasManager
     TerminalIPythonApp
     StoreMagics
     StoreMagics

In [17]: config TerminalInteractiveShell.editor
Out[17]: u'vi'

In [18]: config TerminalInteractiveShell.editor='gedit'

In [19]: config TerminalInteractiveShell.editor
Out[19]: u'gedit'
```

Line Magic commands

- %history = Print input history, with most recent last.
  - -n print line numbers for each input.
  - -o also print outputs for each input.
  - -g show full saved history
  - -g PATTERN search pattern in full history
  - -l ‘n’ = get the last n lines from all sessions. (the default is the last 10 lines)

  - **Exercise**
    - 4 = Line 4, current session
    - 4-6 = Lines 4-6, current session
    - 243/1-5 = Lines 1-5, session 243
    - ~2/7 = Line 7, session 2 before current
    - ~8/1-~6/5 = From the first line of 8 sessions ago, to the fifth line of 6 sessions ago.
    - -g [0-9]{2}

Line Magic commands

Magic commands that can use the historic inputs with the same syntax as the %history magic:

- %macro = Define a macro for future re-execution.
- **%**rerun **=** Re-run previous input
- %save = save input
- %edit = Bring up an editor and execute the resulting code

Line Magic commands

- %macro = Define a macro for future re-execution.

```
In [24]: x=1

In [25]: y=2

In [26]: print(x+y)
3

In [27]: macro my_first_macro 24-26
Macro `my_first_macro` created. To execute, type its name (without quotes).
=== Macro contents: ===
x=1
y=2
print(x+y)

In [28]: my_first_macro
3
```

- %rerun = Re-run previous input
  - -l <n> : Repeat last n lines of input, not including the current.
  - -g foo : Repeat the most recent line which contains foo
- %save = save input

```
In [29]: rerun 24-26
=== Executing: ===
x=1
y=2
print(x+y)
=== Output: ===
3
```

Line Magic commands

```
In [64]: x=3; y=5

In [65]: print x+y
8

In [66]: edit 64-65
IPython will make a temporary file named: /tmp/ipyt
Editing... done. Executing edited code...
8
Out[66]: 'x=3; y=5\nprint x+y'
```

- %edit = Bring up an editor and execute the resulting code
  - -n open the editor at a specified line number
  - -x do not execute the edited code immediately upon exit.

```
In [61]: def x_plus(x_in):
   ....:     return x_in+1
   ....:

In [62]: edit 61
IPython will make a temporary file named: /tmp/ipython_edit_fbg2xu/ipython_edit_Y5qXbT.py
Editing...
(gedit:12308): Gtk-WARNING **: Calling Inhibit failed: GDBus.Error:org.freedesktop.DBus.Erro
onManager was not provided by any .service files
done. Executing edited code...
Out[62]: 'def x_plus(x_in):\n    return x_in+1\n'

In [63]: edit x_plus
Editing... done. Executing edited code...
```

Line Magic commands

- %notebook = This function can export the current IPython history to a notebook file.
  - For example, to export the history to “foo.ipynb” do “%notebook -e foo.ipynb”.
  - To export the history to “foo.py” do “%notebook -e foo.py”.
- %time = Time execution of a Python statement or expression.
- %timeit = Time execution of a Python statement or expression with more options
- %alias = Define an alias for a system command

```
In [98]: alias show echo

In [99]: show take this
take this
```

Line Magic commands

- %who = Print all interactive variables, with some minimal formatting.
- %whos = Like %who, but gives some extra information about each variable.
- %who_ls = Return a sorted list of all interactive variables

- %xdel = Delete a variable

- %reset = Resets the namespace by removing all names defined by the user
- %reset_selective = Resets the namespace by removing names defined by the user.

- %psearch = Search for object in namespaces by wildcard.

Line Magic commands

- %who = Print all interactive variables, with some minimal formatting.
  - If any arguments are given, only variables whose type matches one of these are printed.

```
In [104]: who
abs      func    my_first_macro  my_first_macrto        simple_f       x      x_plus  y      z


In [105]: who int
x        y       z

In [106]: who int function
abs      func    simple_f        x       x_plus  y       z

In [107]: who function str
abs      func    simple_f        x_plus
```

  - always excludes executed names loaded through your configuration file and things which are internal to IPython.
  - This is deliberate, as typically you may load many modules and the purpose of %who is to show you only what you've manually defined.

Line Magic commands

- %whos = Like %who, but gives some extra information about each variable.
- %who_ls = Return a sorted list of all interactive variables

```
In [114]: whos
Variable           Type        Data/Info
-------------------------------------
abs                function    <function abs at 0x7fcd31d495f0>
func               function    <function func at 0x7fcd31d49aa0>
my_first_macro     Macro       x=1\ny=2\nprint(x+y)\n
my_first_macrto    Macro       x=1\ny=2\nz=x+y\nprint z\n
simple_f           function    <function simple_f at 0x7fcd31d496e0>
x                  int         3
x_plus             function    <function x_plus at 0x7fcd31d49ed8>
y                  int         5
z                  int         3

In [115]: whos Macro
Variable           Type     Data/Info
---------------------------------
my_first_macro     Macro    x=1\ny=2\nprint(x+y)\n
my_first_macrto    Macro    x=1\ny=2\nz=x+y\nprint z\n
```

```
In [108]: who_ls
Out[108]:
['abs',
 'func',
 u'my_first_macro',
 u'my_first_macrto',
 'simple_f',
 'x',
 'x_plus',
 'y',
 'z']

In [109]: who_ls int
Out[109]: ['x', 'y', 'z']
```

Line Magic commands

- %xdel = Delete a variable

```
In [134]: k=5

In [135]: who int
b        k        y

In [136]: del k

In [137]: who int
b        y
```

- %reset = Resets the namespace by removing all names defined by the user

```
In [138]: reset
Once deleted, variables cannot be recovered. Proceed (y/[n])? y

In [139]: who
Interactive namespace is empty.
```

  - -s : ‘Soft’ reset: Only clears your namespace, leaving history intact.
  - in/ out : reset input/output history
  - dhist : reset directory history

Line Magic commands

- %reset_selective = Resets the namespace by removing names defined by the user.

```
In [177]: whos
Variable        Type        Data/Info
-------------------------------------
jfg             float       124124.343
new_macro       Macro       jfg= 124124.343\n
this_and_that   function    <function this_and_that at 0x7fcd340938c0>
x               int         12
x3              int         18
x_test          str         abc
z               int         124124

In [178]: reset_selective -f x

In [179]: whos
Variable        Type        Data/Info
-------------------------------------
jfg             float       124124.343
new_macro       Macro       jfg= 124124.343\n
this_and_that   function    <function this_and_that at 0x7fcd340938c0>
z               int         124124
```

Line Magic commands

- %psearch = Search for object in namespaces by wildcard.

```
In [202]: whos
Variable        Type        Data/Info
-----------------------------------
jfg             float       124124.343
new_macro       Macro       jfg= 124124.343\n
this_and_that   function    <function this_and_that at 0x7fcd340938c0>
x               int         1
x3              int         18
x_test          str         abc
z               int         124124

In [203]: psearch x*
x
x3
x_test
xrange

In [204]: psearch -e builtin x*
x
x3
x_test

In [205]: psearch *and* function
this_and_that
```

Line Magic commands

- capturing the output of shell command

```
In [211]: a= ! ls *py

In [212]: a
Out[212]: ['hello_world.py', 'say_hello.py.py', 'some_module.py']

In [213]: b=!cat hello_world.py
```

- Which type is the output?
- What methods does it have?

```
In [220]: type(a)
Out[220]: IPython.utils.text.SList

In [221]: a.
a.append     a.fields     a.get_paths  a.index      a.list       a.p          a.remove     a.sort
a.count      a.get_list   a.get_spstr  a.insert     a.n          a.paths      a.reverse    a.spstr
a.extend     a.get_nlstr  a.grep       a.l          a.nlstr      a.pop        a.s

In [221]: a.grep?
```

Line Magic commands

- Can we reuse this for the input of another magic command?

- The output capture has the following special attributes:
  - .l (or .list) : value as list.
  - .n (or .nlstr): value as newline-separated string.
  - .s (or .spstr): value as space-separated string.

```
In [230]: a.s
Out[230]: 'hello_world.py say_hello.py.py some_module.py'

In [231]: !wc -l $a.s
 12 hello_world.py
  5 say_hello.py.py
  9 some_module.py
 26 total
```

CellMagic commands

- they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument.

```
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%latex  %%perl  %%p
run  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time
  %%timeit  %%writefile
```

```
In [237]: %%bash
   .....: ls -l
   .....: grep -i "hello" *.py
   .....:
total 180
-rw-rw-r--. 1 dsc dsc    95 Apr  4 12:35 hello_world.py
-rw-rw-r--. 1 dsc dsc   212 Apr  6 08:09 say_hello.py.py
-rw-rw-r--. 1 dsc dsc    84 Apr  4 14:34 some_module.py
-rw-rw-r--. 1 dsc dsc   362 Apr  4 14:36 some_module.pyc
-rw-rw-r--. 1 dsc dsc 43675 Apr  6 10:15 test01.ipynb
-rw-rw-r--. 1 dsc dsc  5422 Apr  5 20:17 test03.ipynb
-rw-rw-r--. 1 dsc dsc 89812 Apr  5 22:50 test2.ipynb
-rw-rw-r--. 1 dsc dsc 20208 Apr  5 09:31 test2.ipynb.01
-rw-rw-r--. 1 dsc dsc     0 Apr  5 13:17 test.txt
-rw-rw-r--. 1 dsc dsc   582 Apr  5 22:23 Untitled.ipynb
hello_world.py:print 'Hello world from python!'
say_hello.py.py:def say_hello_then_call_f(f, *args, **kwargs):
say_hello.py.py:        print("Hello! Now I'm going to call this function %s" % f)
```

Functions

- functions are declared using the def keyword and returned from using the return keyword:
- If the end of a function is reached without encountering a return statement, None is returned.

```
In [9]: def my_function(x, y, z=1.5):
   ...:     if z > 1:
   ...:         return z * (x + y)
   ...:     else:
   ...:         return z / (x + y)
   ...:
```

- Each function can have some number of positional arguments and some number of keyword arguments.
- Keyword arguments are most commonly used to specify default values or optional arguments.
- In the above function, x and y are positional arguments while z is a keyword argument.

```
In [10]: my_function(5, 6, z=0.7)
Out[10]: 0.06363636363636363

In [11]: my_function(3.14, 7, 3.5)
Out[11]: 35.49

In [12]: my_function(3.14, 7)
Out[12]: 15.21

In [13]: my_function(5, , z=0.7)
  File "<ipython-input-13-9f59f1d98af2>", line 1
    my_function(5, , z=0.7)
                   ^
SyntaxError: invalid syntax
```

- keyword arguments *must* follow the positional arguments (if any).
- You can specify keyword arguments in any order
- this frees you from having to remember which order the function arguments were specified in

Functions - Namespaces, Scope, and Local Functions

- Functions can access variables in two different scopes: global and local.
- Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed.

```
In [27]: def func():
    ....:     a = []
    ....:     for i in range(5):
    ....:         a.append(i)
    ....:
```

```
In [22]: a = []

In [23]: def func():
    ....:     for i in range(5):
    ....:         a.append(i)
    ....:
```

- In one option by calling func(), the empty list a is created, 5 elements are appended, then a is destroyed when the function exits. In which one?
- Assigning global variables within a function is possible, but they must be declared as global using the global keyword

```
In [40]: a = None

In [41]: def bind_a_variable():
    ....:     global a
    ....:     a=[1,2]
    ....:

In [42]: bind_a_variable(); print(a)
[1, 2]
```

Functions - Namespaces, Scope, and Local Functions

- Functions can be declared anywhere, and there is no problem with having local functions

```
In [43]: def outer_function(x, y, z):
    ....:     def inner_function(a, b, c):
    ....:         pass
    ....:     pass
    ....:
```

- the inner_function will not exist until outer_function is called. Aa soon as outer_function is done executing, the inner_function is destroyed.

- pass is the “no-op” statement in Python. It can be used in blocks where no action is to be taken
- It’s common to use pass as a place-holder in code while working on a new piece of functionality

Functions - Returning Multiple Values

```
In [6]: def f():
    a = 5
    b = 6
    c = 7
    return a, b, c
   ...:

In [7]: d, e, g = f() ;

In [8]: return_value=f()
```

- the function is actually just returning *one* object which is then being unpacked into the result variables.
- A potentially attractive alternative to returning multiple values like above might be to return a dict instead:

```
In [9]: def f():
   ...:     a = 5
   ...:     b = 6
   ...:     c = 7
   ...:     return {'a' : a, 'b' : b, 'c' : c}
   ...:

In [10]: return_value=f()

In [11]: return_value
Out[11]: {'a': 5, 'b': 6, 'c': 7}
```

Functions - Are Objects

- function is used as argument to other function
- ops has a list of the operations to apply to a particular set of values

```
In [12]: def add_one(value):
    ....:     return value+1
    ....:

In [13]: def double_value(value):
    ....:     return value*2
    ....:

In [14]: def add_three(value):
    ....:     return value+3
    ....:

In [15]: math_ops = [add_one, double_value, add_three]
```

```
In [17]: def math_values(values, ops):
    result = []
    for value in values:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
    ....:

In [18]: k=[1,2,3]

In [19]: math_values(k, math_ops)
Out[19]: [7, 9, 11]
```

- map is built in function which applies a function to a collection of some kind

```
In [21]: k
Out[21]: [1, 2, 3]

In [22]: map(add_one, k)
Out[22]: [2, 3, 4]
```

Functions - Anonymous Functions

- or *lambda* functions
- Just simple functions consisting of a single statement, the result of which is the return value.
- defined using the lambda keyword, which has no meaning other than “we are declaring an anonymous function.”

```
In [24]: def short_function(x):
    ....:     return x * 2
    ....:

In [25]: equiv_anon = lambda x: x * 2
```

- They are especially convenient in data analysis because, there are many cases where data transformation functions will take functions as arguments (as we have seen in the previous slide).

```
In [28]: def apply_to_list(some_list, f):
    ....:     return [f(x) for x in some_list]
    ....:

In [29]: ints = [4, 0, 1, 5, 6]

In [30]: apply_to_list(ints, lambda x: x * 2)
Out[30]: [8, 0, 2, 10, 12]
```

Functions - Extended Call Syntax

- with *args, **kwargs
- When you write **func(a, b, c, d=some, e=value**), the positional and keyword arguments are packed up into a tuple and dict, respectively. So the internal function receives a tuple args and dict kwargs and internally does the equivalent of:

  a, b, c = args

  d = kwargs.get('d', d_default_value)

  e = kwargs.get('e', e_default_value)

- it also allows you to specify some of the positional arguments as keywords also (even if they aren't keyword in the function declaration!)

```
In [34]: def say_hello_then_call_f(f, *args, **kwargs):
   ....:     print 'args is', args
   ....:     print 'kwargs is', kwargs
   ....:     print("Hello! Now I'm going to call %s" % f)
   ....:     return f(*args, **kwargs)
   ....:

In [35]: def g(x, y, z=1):
   ....:     return (x + y) / z
```

```
In [36]: say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x7f95fec0c5f0>
Out[36]: 0.6
```

Flow Control – if -else

```
In [262]: x=-1

In [263]: if x < 0:
     ...:     print 'It\'s negative'
     ...:
It's negative
```

```
In [261]: if x < 0:
     ...:     print 'It\'s negative'
     ...: elif x == 0:
     ...:     print 'Equal to zero'
     ...: elif 0 < x < 5:
     ...:     print 'Positive but smaller than 5'
     ...: else:
     ...:     print 'Positive and larger than or equal to 5'
     ...:
Positive and larger than or equal to 5
```

```
In [265]: a = 5; b = 7

In [266]: c = 8; d = 4

In [267]: if a < b or c > d:
     ...:     print "thats it"
     ...:
thats it
```

- the comparison c > d never gets evaluated because the first comparison was True.

Flow Control – for loop

- A for loop can be advanced to the next iteration, skipping the remainder of the block, using the continue keyword.

```
In [269]: sequence = [1, 2, None, 4, None, 5]

In [270]: total = 0

In [271]: for value in sequence:
   .....:     if value is None:
   .....:         continue
   .....:     total += value
   .....:

In [272]: total
Out[272]: 12
```

- A for loop can be exited altogether using the break keyword

```
In [273]: sequence = [1, 2, 0, 4, 6, 5, 2, 1]

In [274]: total_until_5 = 0

In [275]: for value in sequence:
   .....:     if value == 5:
   .....:         break
   .....:     total_until_5 += value
   .....:

In [276]: total
Out[276]: 12
```

Flow Control – while loop

- A while loop specifies a condition and a block of code that is to be executed until the condition evaluates to False or the loop is explicitly ended with break

```
In [277]: x = 256

In [278]: total = 0

In [279]: while x > 0:
   .....:     if total > 500:
   .....:         break
   .....:     total += x
   .....:     x = x // 2
   .....:

In [280]: x
Out[280]: 4

In [281]: total
Out[281]: 504
```

Flow Control – pass

- pass is the "no-op" statement in Python.
- It can be used in blocks where no action is to be taken;
- it is only required because Python uses whitespace to delimit blocks:

```
In [285]: if x < 0:
   .....:     print 'negative!'
   .....: elif x == 0:
   .....:     # TODO: put something smart here
   .....:     pass
   .....: else:
   .....:     print 'positive!'
   .....:
positive!
```

- It's common to use pass as a place-holder in code while working on a new piece of functionality:

```
In [289]: def f(x, y, z):
    # TODO: implement this function!
    pass
   .....:

In [290]: whos function
Variable        Type        Data/Info
-----------------------------------
f               function    <function f at 0x7fcd31d21f50>
this_and_that   function    <function this_and_that at 0x7fcd340938c0>
```

Quick Exercises 1

1. Go to ~/Data/opentraveldata/ and list the files. Repeat the steps for /home/dsc/Data/us_dot/otp and ~/Data/us_dot/traffic/. Use the list of visited directories from dhist towrite for loop which will list the files in every directory with –l option.
2. Implement a function that takes as input three variables, and returns the largest of the three. Do this without using the Python max() function! Make 1 version without any local variable and another with one local variable. (hint: it might be easier to use edit)
3. Write a function that gets 3 inputs: list of numbers, number, number. If the number exists in list it returns its value, otherwise it returns False. Write the function with edit magic command. Recollect the return values as one object and as variables. (hint : use **in** method)
4. For a sequence [1, 2, 3, 4, 5, 6, 7, 8] get the squared values using the lambda function.
5. Write a function that will take Name, and year of birth as inputs, check if year of birth is not int and cast it to int in that case, and print name together with text explaining when the person is to have 100 years (use isinstance)

   centenario(Antonio, 1967)

   Antonio will reach 100 years in 2067.
6. Create a function that accepts string as search string and returns number of lines with this string in a command history (hint : use a in b)

Handling Files

- Most of time we use high-level tools like pandas.read_csv to read data files from disk into Python data structures.
- However, it's important to understand the basics of how to work with files in Python.
- open(path) - by default, the file is opened in read-only mode 'r'.

```
In [390]: path='Finn.txt'

In [391]: f=open(path)

In [392]: f2=open('abx.txt','w')

In [393]: f.
f.close       f.fileno      f.name        f.readinto    f.softspace   f.writelines
f.closed      f.flush       f.newlines    f.readline    f.tell        f.xreadlines
f.encoding    f.isatty      f.next        f.readlines   f.truncate
f.errors      f.mode        f.read        f.seek        f.write
```

- **We can then treat the file handle f like a list and iterate over the lines!**

```
In [394]: for lines in f:
   .....:     #TODO make smth
   .....:     pass
   .....:
```

curl -s http://www.gutenberg.org/cache/epub/76/pg76.txt > Finn.txt

Handling Files

- fopen modes:
  - r Read-only mode
  - w Write-only mode. Creates a new file (deleting any file with the same name)
  - a Append to existing file (create it if it does not exist)
  - r+ Read and write
  - b Add to mode for binary files, that is 'rb' or 'wb'
- file handler methods
  - read([size]) Return data from file as a string, with optional size argument indicating the number of bytes to read
  - readlines([size]) Return list of lines in the file, with optional size argument
  - write(str) Write passed string to file.
  - writelines(strings) Write passed sequence of strings to the file.
  - close() Close the handle
  - flush() Flush the internal I/O buffer to disk
  - seek(pos) Move to indicated file position (integer).
  - tell() Return current file position as integer.
  - closed True if the file is closed.

Quick Exercises 2

1. Open Finn.txt file, read lines into a list. If the list elements have EOL character remove it. Write the list with no EOL characters to the new file. How many lines does the new file have? (hint: empty list is made with [ ])
2. Clear namespace leaving only variable with Finn.txt path and list with lines from the previous task.
3. Create a new version of Finn_nbl.txt with no blank lines using the list with loaded lines from 1 excercise. (we are going to use this file later…)

Scalar Types

- Main types:
  - int = Signed integer with maximum value determined by the platform.
  - long = Arbitrary precision signed integer. Large int values are automatically converted to long.
  - float = Double-precision (64-bit) floating point number. Note there is no separate double type.
  - str = String type
  - bool = A True or False value
  - None = The Python “null” value (only one instance of the None object exists)

Scalar Types - Numeric types

- The size of the integer which can be stored as an int is dependent on your platform (whether 32 or 64-bit), but Python will transparently convert a very large integer to long, which can store arbitrarily large integers.

```
In [521]: k=1423432

In [522]: k, type(k)
Out[522]: (1423432, int)

In [523]: k=12345678987987999342323

In [524]: k, type(k)
Out[524]: (12345678987987999342323L, long)

In [525]: k ** 19
Out[525]: 54800513125323773345251270381327312618370811021516443407285245980365126689197255516690150872085837363502190286655299971487823909738183098704204866549737436395629386393404477079362848446863271473432919315547193774826533364099725977658262742073732753769499763425362209369527815800889282483805755640877410037879421767536678719413015706339426325425071384576180861857832903916645271755667341759562604967994409470274108543787L
```

```
In [527]: k.
k.bit_length   k.conjugate    k.denominator  k.imag         k.numerator    k.real

In [527]: k.bit_length()
Out[527]: 74
```

Scalar Types - Numeric types

- Floats…nder the hood each one is a double-precision (64 bits) value. They can also be expressed using scientific notation.

```
In [536]: fval = 7.243

In [537]: fval2 = 6.78e-5

In [538]: 3 / 2
Out[538]: 1

In [539]: 3 / float(2)
Out[539]: 1.5
```

- In Python 3, integer division not resulting in a whole number will always yield a floating point number

```
In [542]: from __future__ import division

In [543]: 3/2
Out[543]: 1.5

In [544]: fval.
fval.as_integer_ratio  fval.fromhex           fval.imag              fval.real
fval.conjugate         fval.hex               fval.is_integer
```

Scalar Types - Numeric types

- Complex numbers are written using j for the imaginary part:

```
In [550]: cval = 1 + 2j

In [551]: (cval * (1 - 2j))*10
Out[551]: (50+0j)

In [552]: type(cval)
Out[552]: complex

In [553]: real_val=cval.real

In [554]: cval.
cval.conjugate  cval.imag       cval.real
```

Scalar Types – Strings

- You can write string literal using either single quotes ' or double quotes "
- For multiline strings with line breaks, you can use triple quotes, either ''' or """

```
In [557]: a = 'one way of writing a string'

In [558]: b = "another way"

In [559]: c = """
   .....: This is a longer string that
   .....: spans multiple lines
   .....: """
```

- strings are immutable; you cannot modify a string without creating a new string

```
In [561]: a="just another string"

In [562]: a[5]='f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-562-308e8794badb> in <module>()
----> 1 a[5]='f'

TypeError: 'str' object does not support item assignment
```

Scalar Types – Strings

- you cannot modify a string without creating a new string

```
In [563]: a
Out[563]: 'just another string'

In [564]: b=a.replace('another','changed this')

In [565]: b
Out[565]: 'just changed this string'
```

```
In [563]: a.
a.capitalize  a.find        a.isspace     a.partition   a.rstrip      a.translate
a.center      a.format      a.istitle     a.replace     a.split       a.upper
a.count       a.index       a.isupper     a.rfind       a.splitlines  a.zfill
a.decode      a.isalnum     a.join        a.rindex      a.startswith
a.encode      a.isalpha     a.ljust       a.rjust       a.strip
a.endswith    a.isdigit     a.lower       a.rpartition  a.swapcase
a.expandtabs  a.islower     a.lstrip      a.rsplit      a.title
```

Scalar Types – Strings

find = Return the lowest index where the substring is found
index = Like find() but raise ValueError when the substring is not found.

```
In [570]: a
Out[570]: 'just another string'

In [571]: a.find('o')
Out[571]: 7

In [572]: a.find('oth')
Out[572]: 7

In [573]: a.find('st')
Out[573]: 2

In [574]: a.find('sdsd')
Out[574]: -1

In [575]: a.index('sdsd')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-575-1bd6af0a122d> in <module>()
----> 1 a.index('sdsd')

ValueError: substring not found
```

Scalar Types – Strings

- capitalize
- upper/lower
- title

```
In [578]: a
Out[578]: 'just another string'

In [579]: a.capitalize()
Out[579]: 'Just another string'

In [580]: a.upper()
Out[580]: 'JUST ANOTHER STRING'

In [581]: a.title()
Out[581]: 'Just Another String'
```

- count

```
In [582]: a.count('o')
Out[582]: 1

In [583]: a.count('r')
Out[583]: 2
```

- isdigit

```
In [592]: '5.4e-5'.isdigit()
Out[592]: False

In [593]: '23.5'.isdigit()
Out[593]: False

In [594]: '2'.isdigit()
Out[594]: True
```

Scalar Types – Strings

- split

```
In [596]: a
Out[596]: 'just another string'

In [597]: a.split(" ")
Out[597]: ['just', 'another', 'string']

In [598]: a.split("t")
Out[598]: ['jus', ' ano', 'her s', 'ring']
```

- splitlines

```
In [601]: z="first line \r\n  continues to secod \n which might not be seen \r\n"

In [602]: z.splitlines()
Out[602]: ['first line ', '  continues to secod ', ' which might not be seen ']
```

- S.join(iterable) = Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

```
In [622]: a='just another string'

In [623]: "^".join(a)
Out[623]: 'j^u^s^t^ ^a^n^o^t^h^e^r^ ^s^t^r^i^n^g'
```

Scalar Types – Strings

- Strings are a sequence of characters and therefore can be treated like other sequences, such as lists and tuples

```
In [628]: s="Python is WOW"

In [629]: list(s)
Out[629]: ['P', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 'W', 'O', 'W']
```

- The backslash character \ is an escape character, meaning that it is used to specify special characters like newline \n or unicode characters. To write a string literal with backslashes, you need to escape them or preface the leading quote of the string with r.

```
In [644]: s = '12\\34\n56'

In [645]: s
Out[645]: '12\\34\n56'

In [646]: print s
12\34
56

In [647]: s2=r'this\has\no\special\characters'

In [648]: s2
Out[648]: 'this\\has\\no\\special\\characters'

In [649]: print s2
this\has\no\special\characters
```

Scalar Types – Strings

- Adding two strings together concatenates them and produces a new string

```
In [650]: a = 'this is the first half '

In [651]: b= 'and this is the second half'

In [652]: a+b
Out[652]: 'this is the first half and this is the second half'
```

- Strings with a % followed by one or more format characters is a target for inserting a value into that string.

```
In [656]: template = '%.2f %s are worth $%d'

In [657]: template % (7.3560, 'Croatia Kunas', 1)
Out[657]: '7.36 Croatia Kunas are worth $1'

In [658]: print '%.2f %s are worth $%d' % (7.3560, 'Croatia Kunas', 1)
7.36 Croatia Kunas are worth $1
```

- Many Python objects can be converted to a string using the str function

```
In [660]: a="345"

In [661]: str(a)
Out[661]: '345'
```

Scalar Types – Booleans

- Boolean values are combined with the and and or keywords:
- Most objects in Python have a notion of true- or falseness.
- For example, empty sequences lists, dicts, tuples, etc.) are treated as False if used in control flow (as above with the empty list b).

```
In [665]: a=[]

In [666]: if a:
   .....:     print "A is not empty"
   .....: else:
   .....:     print "aaaaa... it is empty"
   .....:
aaaaa... it is empty

In [667]: (True and True) and (True or False)
Out[667]: True
```

- You can see exactly what boolean value an object has by invoking bool on it:

```
In [668]: bool([]), bool([1, 2, 3])
Out[668]: (False, True)

In [669]: bool([]), bool([1, 2, 3])
Out[669]: (False, True)
```

Scalar Types – None

- None is the Python null value type. If a function does not explicitly return a value, it implicitly returns None.

```
In [670]: a=None; b=5

In [671]: a is None, b is not None
Out[671]: (True, True)
```

- None is also a common default value for optional function arguments

```
In [673]: def add_and_maybe_multiply(a, b, c=None):
   .....:     result = a + b
   .....:     if c is not None:
   .....:         result = result * c
   .....:     return result
   .....:

In [674]: add_and_maybe_multiply(3,4)
Out[674]: 7

In [675]: add_and_maybe_multiply(3,4,2)
Out[675]: 14
```

Scalar Types – Type casting

- The str, bool, int and float types are also functions which can be used to cast values to those types

```
In [687]: s = '3.14159'

In [688]: fval = float(s)

In [689]: type(fval)
Out[689]: float

In [690]: int(fval)
Out[690]: 3

In [691]: bool(fval)
Out[691]: True

In [692]: bool('0')
Out[692]: True

In [693]: bool(0)
Out[693]: False
```

Quick Excercises 3

1. Obtain the difference in number of lines between original Finn file and Finn file without blank lines and print the result .
2. Prepare a list with 10 names. Make a code that will put all vowels to capitals and every other character to lower letters.

Data structures

- tuple: one-dimensional, **fixed-length**, immutable sequence of Python objects (the objects CAN be mutable!!!)
- lists: one-dimensional, **variable-length**, mutable sequence of Python objects
- dict: **flexibly-sized** collection of key-value pairs, where key and value are Python objects
- set: **unordered** collection of unique elements (like dicts, but keys only, no values)

Data structures - Tuples

- tuple: one-dimensional, **fixed-length**, immutable sequence of Python objects (the objects CAN be mutable!!!)
- creation:
  - with a comma-separated sequence of values
  - by converting **any sequence or iterator** by invoking tuple()
  - ( ) = empty tuple
- can be nested

```
In [712]: tup = 4, 5, 6

In [713]: tup
Out[713]: (4, 5, 6)

In [714]: tup = tuple('string')

In [715]: tup
Out[715]: ('s', 't', 'r', 'i', 'n', 'g')

In [716]: tuple([4, 0, 2])
Out[716]: (4, 0, 2)

In [717]: nested_tup = (4, 5, 6), (7, 8), ('A',8,'abcd');

In [718]: nested_tup
Out[718]: ((4, 5, 6), (7, 8), ('A', 8, 'abcd'))
```

Data structures - Tuples

- elements can be accessed with square brackets [ ]
- sequences are 0-indexed

```
In [744]: tup = tuple('string')

In [745]: tup[0]
Out[745]: 's'

In [746]: tup[:3]
Out[746]: ('s', 't', 'r')

In [747]: tup[2:4]
Out[747]: ('r', 'i')

In [748]: tup[4:]
Out[748]: ('n', 'g')

In [749]: tup[1:5:2]
Out[749]: ('t', 'i')
```

- Negative indices = slice the sequence relative to the end
- Negative step = reverse the direction of moving through

```
In [780]: tup[0:]
Out[780]: ('s', 't', 'r', 'i', 'n', 'g')

In [781]: tup[-1:]
Out[781]: ('g',)

In [782]: tup[-2:]
Out[782]: ('n', 'g')

In [783]: tup[::-1]
Out[783]: ('g', 'n', 'i', 'r', 't', 's')

In [784]: tup[::-2]
Out[784]: ('g', 'i', 't')

In [785]: tup[-2::1]
Out[785]: ('n', 'g')

In [786]: tup[-2::-1]
Out[786]: ('n', 'i', 'r', 't', 's')
```

Data structures - Tuples

- **In tuple it is not possible to modify the position of object**
- But… the objects stored in a tuple may be mutable themselves, once created!!!

```
In [792]: tup = tuple(['foo', [1, 2], True])

In [793]: tup[2]=False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-793-b2aa2cf1b676> in <module>()
----> 1 tup[2]=False

TypeError: 'tuple' object does not support item assignment

In [794]: tup[3]=123
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-794-e354dbc1d7ea> in <module>()
```

```
In [808]: tup = tuple(['foo', [1, 2], True])

In [809]: tup[1].append(23)

In [810]: tup[1].insert(1,14)

In [811]: tup
Out[811]: ('foo', [1, 14, 2, 23], True)
```

Data structures - Tuples

- Tuples can be concatenated using the + operator to produce longer tuples

```
In [847]: tup = tuple(['foo', [1, 2], True])

In [848]: tup = tup + tuple([23,45])+tuple([[23,45]])+tuple('Askme')+tuple(['Answer'])

In [849]: tup
Out[849]: ('foo', [1, 2], True, 23, 45, [23, 45], 'A', 's', 'k', 'm', 'e', 'Answer')

In [850]: tup +=tuple([True])
```

- Multiplying a tuple by an integer, has the effect of concatenating together that many copies of the tuple.

```
In [853]: tup
Out[853]: ('foo', [1, 2], True)

In [854]: tup *2
Out[854]: ('foo', [1, 2], True, 'foo', [1, 2], True)
```

Data structures - Tuples

- Be careful when creating tuples the objects themselves are not copied, only the references to them.

```
In [871]: a[0]=4; b.append(-23); c=7; d[0]='yes'; e='aa5'

In [872]: a= [1,2,3]; b=[23]; c=50; d=['Txt']; e='a'

In [873]: tup=tuple([a,b,c,d,e])

In [874]: tup2=tup*2

In [875]: a[0]=4; b.append(-23); c=7; d[0]='yes'; e='aa5'

In [876]: tup2
Out[876]:
([4, 2, 3],
 [23, -23],
 50,
 ['yes'],
 'a',
 [4, 2, 3],
 [23, -23],
 50,
 ['yes'],
 'a')
```

Data structures - Tuples

- Unpacking tuples

```
In [929]: tup = (4, 5, 6)

In [930]: a, b, c = tup

In [931]: b
Out[931]: 5

In [932]: tup = 4, 5, (6, 7)

In [933]: a, b, (c, d) = tup

In [934]: d
Out[934]: 7

In [935]: a, b, cd = tup

In [936]: cd
Out[936]: (6, 7)
```

- Using this functionality it's easy to swap variable names

```
In [940]: a,b
Out[940]: (5, 4)

In [941]: a,b=b,a

In [942]: a,b
Out[942]: (4, 5)
```

- common use of variable unpacking when iterating over sequences of tuples (or lists):

```
In [943]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [944]: for a, b, c in seq:
   .....:         print a,b,c
   .....:
1 2 3
4 5 6
7 8 9
```

Data structures - Tuples

- Methods :
  - index() = return first index of value
  - count() = counts the number of occurrences of a value

```
In [956]: a = (1, 2, 2, 2, 3, 4, 2)

In [957]: a.count(2)
Out[957]: 4

In [958]: a.index(2)
Out[958]: 1

In [959]: 565 in a
Out[959]: False

In [960]: b=('this', 'is', 'my', 'home')

In [961]: b.count('i')
Out[961]: 0

In [962]: b.count('is')
Out[962]: 1

In [963]: b.count('isd')
Out[963]: 0

In [964]: b.index('isd')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
```

Data structures - Lists

- lists: one-dimensional, **variable-length**, mutable sequence of Python objects

- creation:
  - using square brackets [ ]
  - by converting **any sequence or iterator** by invoking list()
  - [ ] = empty list

- can be nested

- **Lists and tuples** are semantically similar as one-dimensional sequences of objects and thus **can be used interchangeably in many functions**

```
In [1030]: a_list = [2, 3, 7, None]

In [1031]: tup1 = ('ich', 'bin')

In [1032]: tup2 = ('du', 'bist')

In [1033]: b_list1=list(tup1)

In [1034]: b_list1
Out[1034]: ['ich', 'bin']

In [1035]: b_list12a=list([tup1,tup2])

In [1036]: b_list12a
Out[1036]: [('ich', 'bin'), ('du', 'bist')]

In [1037]: b_list12b=list(tup1+tup2)

In [1038]: b_list12b
Out[1038]: ['ich', 'bin', 'du', 'bist']

In [1039]: b_list12c=list([list(tup1),list(tup2)])

In [1040]: b_list12c
Out[1040]: [['ich', 'bin'], ['du', 'bist']]

In [1041]: len(b_list12a), len(b_list12b), len(b_list12c)
Out[1041]: (2, 4, 2)
```

Data structures - Lists

- Adding and removing elements
  - append(S) = add element S at the end
  - extend([ ]) = append multiple elements
  - insert(N,S) = insert element S at position N
  - remove(S) = removes the first S from the list
  - pop(N) = remove and return element at position N

```
In [1080]: a=['I','live', 'in']

In [1081]: a.append('Madrid')

In [1082]: a=a+['since','2012']

In [1083]: a
Out[1083]: ['I', 'live', 'in', 'Madrid', 'since', '2012']

In [1084]: a.pop(2);a.pop(2);

In [1085]: a
Out[1085]: ['I', 'live', 'since', '2012']

In [1086]: a.insert(2,'here')

In [1091]: a
Out[1091]: ['I', 'live', 'here', 'since', '2012', 'here']

In [1092]: a.remove('here')

In [1093]: a
Out[1093]: ['I', 'live', 'since', '2012', 'here']

In [1094]: 'test' in a
Out[1094]: False

In [1095]: a.extend(['in','Madrid.'])

In [1096]: a
Out[1096]: ['I', 'live', 'since', '2012', 'here', 'in', 'Madrid.']
```

- insert is computationally expensive compared with append as references to subsequent elements have to be shifted internally to make room for the new element.

Data structures - Lists

- List concatenation (with +) is a more expensive operation then extend() since a new list must be created and the objects copied over.

```
everything = [ ]
for chunk in list_of_lists:
    everything.extend(chunk)
```

Vs.

```
everything = [ ]
for chunk in list_of_lists:
    everything = everything + chunk
```

- Using extend to append elements to an existing list is preferable especially if you are building up a large list!!!

Data structures - Lists

- reverse() = reverses objects of list in place
- sort (key=method, reverse=True/False) = in-place sorting based on key method

```
In [1168]: a = [7, 2, 5, 1, 3]

In [1169]: a.sort(); a
Out[1169]: [1, 2, 3, 5, 7]

In [1170]: b = ['Hello', 'small', 'helll', 'foxes', 'he', 'Man']

In [1171]: b.sort(); b
Out[1171]: ['Hello', 'Man', 'foxes', 'he', 'helll', 'small']

In [1172]: b.sort(key=str); b
Out[1172]: ['Hello', 'Man', 'foxes', 'he', 'helll', 'small']

In [1173]: b.sort(key=str.lower);b
Out[1173]: ['foxes', 'he', 'helll', 'Hello', 'Man', 'small']

In [1174]: b.sort(key=len,reverse=True);b
Out[1174]: ['foxes', 'helll', 'Hello', 'small', 'Man', 'he']

In [1175]: b.sort(key=lambda x:x.count('l'));b
Out[1175]: ['foxes', 'Man', 'he', 'Hello', 'small', 'helll']
```

- sorted (list, key=method, reverse=True/False)
  - works on any iterable

```
In [1208]: c=sorted(b, key=lambda x:x.count('l'))

In [1209]: c
Out[1209]: ['Madrid', 'in', 'I', 'live']

In [1210]: a=1,24,5,67,7,4,34

In [1211]: sorted(a)
Out[1211]: [1, 4, 5, 7, 24, 34, 67]

In [1219]: sorted('say hy')
Out[1219]: [' ', 'a', 'h', 's', 'y', 'y']
```

We need this to proceed - Enumerate

- It’s common when iterating over a sequence to want to keep track of the index of the current item
- enumerate() returns a sequence of (i, value) tuples

```
In [1319]: counter=0

In [1320]: for value in range(10,20):
   ......:     print counter, value
   ......:     counter +=1
   ......:
0 10
1 11
2 12
3 13
4 14
5 15
6 16
7 17
8 18
9 19
```

```
In [1315]: for i, value in enumerate(range(10,20)):
   ......:     print i, value
   ......:
0 10
1 11
2 12
3 13
4 14
5 15
6 16
7 17
8 18
9 19
```

- useful especially when constructing a dict

- reversed() = iterates over the elements of a sequence in reverse order

We need this to proceed - Zip

- "pairs" up the elements of a number of lists, tuples, or other sequences, to create a list of tuples

```
In [1276]: seq1=1,2,3,4

In [1277]: seq2=[5,6,7,8]

In [1278]: seq3=[True, False, True]

In [1279]: zip(seq1, seq2, seq3)
Out[1279]: [(1, 5, True), (2, 6, False), (3, 7, True)]

In [1280]: for i, (a, b, c) in enumerate(zip(seq1, seq2, seq3)):
   ......:     print('%d: %d, %d %s' % (i, a, b,c))
   ......:
0: 1, 5 True
1: 2, 6 False
2: 3, 7 True
```

- uzip is also done with zip()

```
In [1326]: zipped=zip(seq1, seq2, seq3)

In [1327]: seq1_unzip, seq2_unzip, seq3_unzip=zip(*zipped)

In [1328]: seq1_unzip, seq2_unzip, seq3_unzip
Out[1328]: ((1, 2, 3), (5, 6, 7), (True, False, True))
```

Data structures - Dicts

- dict: **flexibly-sized** collection of key-value pairs, where key and value are Python objects
- A more common name for it is hash map or associative array.

- creation:
  - curly braces { } and using colons **:** to separate keys and values
  - by using dict() method over (key, value) pairs
  - { } = empty dict

```
In [1267]: dict(key_one_A=1, key_two_A=2)
Out[1267]: {'key_one_A': 1, 'key_two_A': 2}

In [1268]: dict((('key_one_B',1),('key_two_B',2)))
Out[1268]: {'key_one_B': 1, 'key_two_B': 2}

In [1269]: map_for_dict=('key_one_B',1),('key_two_B',2)

In [1270]: dict(map_for_dict)
Out[1270]: {'key_one_B': 1, 'key_two_B': 2}

In [1271]: dict(zip(('key_one_C', 'key_two_C'), (1,2)))
Out[1271]: {'key_one_C': 1, 'key_two_C': 2}

In [1272]: keys=('key_one_C', 'key_two_C')

In [1273]: values=(1,2)

In [1274]: zip(keys, values)
Out[1274]: [('key_one_C', 1), ('key_two_C', 2)]

In [1275]: dict(zip(keys, values))
Out[1275]: {'key_one_C': 1, 'key_two_C': 2}
```

Data structures - Dicts

- Elements can be accessed , inserted or set using the same syntax as accessing elements of a list or tuple

```
In [1358]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [1359]: d1['St'] = 'Split'; d1[4]='integer'

In [1360]: d1[4]
Out[1360]: 'integer'

In [1361]: del d1['a']

In [1362]: d1
Out[1362]: {4: 'integer', 'St': 'Split', 'b': [1, 2, 3, 4]}

In [1363]: a=d1.pop('St')

In [1364]: a
Out[1364]: 'Split'

In [1365]: d1
Out[1365]: {4: 'integer', 'b': [1, 2, 3, 4]}
```

Data structures - Dicts

```
In [1381]: d1
Out[1381]: {'a': 'some value', 'b': [1, 2, 3, 4]}

In [1382]: d1.
d1.clear       d1.get         d1.iteritems   d1.keys        d1.setdefault  d1.viewitems
d1.copy        d1.has_key     d1.iterkeys    d1.pop         d1.update      d1.viewkeys
d1.fromkeys    d1.items       d1.itervalues  d1.popitem     d1.values      d1.viewvalues
```

- clear() = Remove all items from dict
- get(S, V) = search for S, and return V if you don't find it
- keys() = lists of the keys
- values() = lists of the values
- update(D) = merged into and overwrite if key already exists

```
In [1384]: d1
Out[1384]: {'a': 'some value', 'b': [1, 2, 3, 4]}

In [1385]: d1.get('Spu','Not inside')
Out[1385]: 'Not inside'

In [1386]: d1.keys(), d1.values()
Out[1386]: (['a', 'b'], ['some value', [1, 2, 3, 4]])

In [1387]: d2={'b':'as you see', 5:'second integer'}

In [1388]: d1.update(d2)

In [1389]: d1
Out[1389]: {5: 'second integer', 'a': 'some value', 'b': 'as you see'}
```

Data structures - Sets

- set: **unordered** collection of unique elements (like dicts, but keys only, no values)
- like dicts, but keys only, no values

- creation:
  - curly braces { } (no colons inside as no keys are present)
  - by using set() method
  - set({ })= empty set

```
In [6]: a=set([2, 2, 2, 1, 3, 3])

In [7]: b={2, 2, 2, 1, 3, 3}

In [8]: type(a), type(b)
Out[8]: (set, set)

In [9]: a
Out[9]: {1, 2, 3}

In [10]: b
Out[10]: {1, 2, 3}

In [11]: c=set({})

In [12]: c
Out[12]: set()
```

Data structures - Sets

- support mathematical *operations* like:
  - a.union(b) = a | b
  - a.intersection(b) = a & b
  - a.difference(b) = a - b
  - a.symmetric_difference(b) = a ^ b

```
In [16]: a = {1, 2, 3, 4, 5}

In [17]: b = {3, 4, 5, 6, 7, 8}

In [18]: a | b # union (or)
Out[18]: {1, 2, 3, 4, 5, 6, 7, 8}

In [19]: a & b # intersection (and)
Out[19]: {3, 4, 5}

In [20]: a - b # difference
Out[20]: {1, 2}

In [21]: a ^ b # symmetric difference (xor)
Out[21]: {1, 2, 6, 7, 8}
```

Data structures - Sets

- You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:
- sets are equal if their contents are equal

```
In [27]: a_set = {1, 2, 3, 4, 5}

In [28]: {3, 2, 1}.issubset(a_set)
Out[28]: True

In [29]: a_set.issuperset({2, 1, 3})
Out[29]: True

In [30]: {1, 2, 3} == {3, 2, 1}
Out[30]: True
```

- Other methods

```
In [14]: c
Out[14]: set()

In [15]: c.
c.add                       c.intersection              c.remove
c.clear                     c.intersection_update       c.symmetric_difference
c.copy                      c.isdisjoint                c.symmetric_difference_update
c.difference                c.issubset                  c.union
c.difference_update         c.issuperset                c.update
c.discard                   c.pop
```

Nice to know– Comprehensions

- List comprehensions allow to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression.

[expr for val in collection if condition] **=**

```
result = [ ]
for val in collection:
    if condition:
        result.append(expr)
```

```
In [51]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [52]: [x.upper() for x in strings if len(x) > 2]
Out[52]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Nice to know– Comprehensions

- Dict and set comprehensions:

dict_comp = {key-expr : value-expr for value in collection if condition}

set_comp = {*expr* for value in collection if *condition*}

```
In [53]: unique_lengths = {len(x) for x in strings}

In [54]: unique_lengths
Out[54]: {1, 2, 3, 4, 6}

In [55]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [56]: loc_mapping
Out[56]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}

In [57]: loc_mapping2 = dict((val, idx) for idx, val in enumerate(strings))

In [58]: loc_mapping2
Out[58]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nice to know– Comprehensions

- nested list comprehensions are a bit hard to wrap your head around.
- The for parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before.
- example where we “flatten” a list of tuples of integers into a simple list of integers:

```
In [63]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [64]: flattened = [x for tup in some_tuples for x in tup]

In [65]: flattened
Out[65]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [66]: flattened2 = []

In [67]: for tup in some_tuples:
   ....:     for x in tup:
   ....:         flattened2.append(x)
   ....:

In [68]: flattened2
Out[68]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Keep in mind that the order of the for expressions would be the same if you wrote a nested for loop instead of a list comprehension

Quick Excercises 4

1. Prepare a list with 10 names. Make a function with input variables: list, and character; that returns a list containing all names with one or more characters ’s in them (names). Do one function first with standard for loop and another with list comprehensions.
2. Sort a collection of strings by the number of distinct letters in each string. (hint: use set)
3. Reverse word order from the input string
4. Categorize a list of words by their first letter (hint: use dict)

Annex – Jupyter Notebook

- Jupyter Notebook
- interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media.

- Go out of iPython, type **ipython notebook** in your terminal

Jupyther notebook

- Jupyther notebook app is a server that appears in your browser at a default address (*http://localhost:8888*).
- Closing the browser will not shut down the server.
- You can reopen the previous address and the Jupyther notebook app will be redisplayed.

Jupyther notebook

- When a notebook is opened, its “computational engine” (called the kernel) is automatically started.
- Closing the notebook browser tab, will not shut down the kernel, instead the kernel will keep running until is explicitly shut down.

Jupyther notebook

- You can run the notebook document step-by-step (one cell a time) by pressing shift + enter.
- You can run the whole notebook in a single step by clicking on the menu Cell -> Run All.
- To restart the kernel (i.e. the computational engine), click on the menu Kernel -> Restart. This can be useful to start over a computation from scratch (e.g. variables are deleted, open files are closed, etc...).

Command Mode (press Esc to enable)

- F: find and replace
- Ctrl-Shift-P: open the command palette
- Enter: enter edit mode
- Shift-Enter: run cell, select below
- Ctrl-Enter: run selected cells
- Alt-Enter: run cell, insert below
- Y: to code
- M: to markdown
- R: to raw
- 1: to heading 1
- 2: to heading 2
- 3: to heading 3
- 4: to heading 4
- 5: to heading 5
- 6: to heading 6
- K: select cell above
- Up: select cell above
- Down: select cell below
- J: select cell below
- Shift-K: extend selected cells above
- Shift-Up: extend selected cells above
- Shift-Down: extend selected cells below
- Shift-J: extend selected cells below
- A: insert cell above
- B: insert cell below
- X: cut cell
- C: copy cell
- Shift-V: paste cell above
- V: paste cell below
- Z: undo cell deletion
- D, D: delete selected cell
- Shift-M: merge selected cells, or current cell with cell below if only one cell selected
- Ctrl-S: Save and Checkpoint
- S: Save and Checkpoint
- L: toggle line numbers
- O: toggle output of selected cells
- Shift-O: toggle output scrolling of selected cells
- H: show keyboard shortcuts
- I, I: interrupt kernel
- 0, 0: restart the kernel (with dialog)
- Esc: close the pager
- Q: close the pager
- Shift-Space: scroll notebook up
- Space: scroll notebook down

Edit Mode (press `Enter` to enable)

`Tab` : code completion or indent

`Shift-Tab` : tooltip

`Ctrl-]` : indent

`Ctrl-[` : dedent

`Ctrl-A` : select all

`Ctrl-Z` : undo

`Ctrl-Shift-Z` : redo

`Ctrl-Y` : redo

`Ctrl-Home` : go to cell start

`Ctrl-Up` : go to cell start

`Ctrl-End` : go to cell end

`Ctrl-Down` : go to cell end

`Ctrl-Left` : go one word left

`Ctrl-Right` : go one word right

`Ctrl-Backspace` : delete word before

`Ctrl-Delete` : delete word after

`Ctrl-M` : command mode

`Ctrl-Shift-P` : open the command palette

`Esc` : command mode

`Shift-Enter` : run cell, select below

`Ctrl-Enter` : run selected cells

`Alt-Enter` : run cell, insert below

`Ctrl-Shift--` : split cell

`Ctrl-Shift-Subtract` : split cell

`Ctrl-S` : Save and Checkpoint

`Down` : move cursor down

`Up` : move cursor up

Annex: The Zen of Python

- by Tim Peters

- Long time Pythoneer
- put Python's design into 20 aphorisms, only 19 of which have been written down.

- type: import this